# ソートのアルゴリズム

## 1　はじめに

与えられた実数の並びを規則に従って並べ替えることをソートと呼びます。ソートのアルゴリズムはすでにさまざまに研究がされており、各種プログラミングですでに実装されています。このノートではアルゴリズムの勉強のためにソートの有名どころのアルゴリズムを実装してみます。

まず、アルゴリズムそのものの説明に入る前にソートとは何かを例で見てみます。例えば、以下の実数の並びが与えられているとします。

$$4,\ 8,\ 1,\ -3,\ 5.1,\ -7.2,\ 10 \tag{1}$$

これを小さな数字順、すなわち左が小さな数字、右が大きな数字になるように並び替えると以下のようになります。

$$-7.2,\ -3,\ 1,\ 4,\ 5.1,\ 8,\ 10 \tag{2}$$

このようにある一定の規則に従って並べ替えることをソートと呼びます。この並べ替えは必ずしも小さな数字順である必要はなく大きな数字から順に小さな数字へと並べていっても構いません。ただし以下ではこのノードではソートというとき小さな数字の順番に並べることを意味するとします。

上にあげた問題は数字が 7 つでしたのでソートを行うのに大した手間ではなかったですが、実際のデータを扱うときにはずっと大きな文字の並びになることもあるでしょう。そのようなときに如何に効率的にソートを行うのかは古くからある重要な問題です。このノートではよく知られているアルゴリズムとしてバブルソート、セレクションソート、インサーションソート、シェルソート、ヒープソート、クイックソート、マージソートについて扱います。それぞれのアルゴリズムについて Python で実装したものを github に挙げています。

## 2　アルゴリズムと実装

それではそれぞれのアルゴリズムの詳細について説明していきます。以下では数字の列の大きさを $n$ とします。

### 2.1　バブルソート

与えられた数字の列に対してソートの後の数字の列の並び合う数字を比較すると左の数字より右の数字が大きいはずです。このことから順番に隣り合う数字を確認していって左の数字が右のより大きければ入れ替えるという操作をすることでソートを実現できそうなことがわかります。この考え方を使ってバブルソートでは、列の先頭から順番に隣り合う数字を比較していき左の数字が右の数字より大きな数字のときには、数字を入れ替えるという作業を行います。この操作を最後まで行うと列の 1 番右側 (n 番目) に一番大きな数字が来ることがわかります。このとき一番右の数字以外はまだソートが完了していま

せん。そこで、次に列の先頭から n-1 番目の数字までを同様に隣同士の大小比較をして数字を並べ替えていきます。以下、順番に繰り返していくことで最終的にソートされた数字の列を得ることができます。

具体的にこのアルゴリズムを見るために以下の数字の列を考えます。

$$5, 4, 8, 1, 3, 2, 7, 6 \tag{3}$$

まず、先頭の数字から隣り合う数字を比較して順番に最後まで数字を並べ替えていくと以下のようになります。

$$4, 5, 8, 1, 3, 2, 7, 6 \tag{4}$$
$$4, 5, 1, 8, 3, 2, 7, 6 \tag{5}$$
$$4, 5, 1, 3, 8, 2, 7, 6 \tag{6}$$
$$4, 5, 1, 3, 2, 8, 7, 6 \tag{7}$$
$$4, 5, 1, 3, 2, 7, 8, 6 \tag{8}$$
$$4, 5, 1, 3, 2, 7, 6, 8 \tag{9}$$

この段階で一番右の数字が、最大の数になっていることがわかります。一方で、他の箇所についてはまだ数字が小さな順に並んでいません。そこで再び先頭から隣り合う数字の大小関係に基づいて数字を並べ替えていきます。

$$4, 5, 1, 3, 2, 7, 6, 8 \tag{10}$$
$$4, 1, 5, 3, 2, 7, 6, 8 \tag{11}$$
$$4, 1, 3, 5, 2, 7, 6, 8 \tag{12}$$
$$4, 1, 3, 2, 5, 7, 6, 8 \tag{13}$$
$$4, 1, 3, 2, 5, 6, 7, 8 \tag{14}$$

これで最後の数字二つはソートされていることがわかります。この手続きを繰り返すと、

$$4, 1, 3, 2, 5, 6, 7, 8 \tag{15}$$
$$1, 4, 3, 2, 5, 6, 7, 8 \tag{16}$$
$$1, 3, 4, 2, 5, 6, 7, 8 \tag{17}$$
$$1, 3, 2, 4, 5, 6, 7, 8 \tag{18}$$
$$1, 2, 3, 4, 5, 6, 7, 8 \tag{19}$$
$$1, 2, 3, 4, 5, 6, 7, 8 \tag{20}$$

となり、最終的に全体がソートされた結果が得られます。このアルゴリズムを疑似コードで書くと以下のようになります。

**Algorithm 1** bubble sort

```
for i = 0; i < n − 1; ++i do
  for j = 0; j < n − i − 1; ++j do
    if data[i] > data[i + 1] then
      data[i] と data[i+1] を入れ替える
    end if
  end for
end for
```

このアルゴリズムは for ループの繰り返しの数から計算量はおおよそ

$$\sum_{i=0}^{n-1}(n-i) \tag{21}$$

となるので、$\mathcal{O}(n^2)$ となる。

## 2.2 セレクションソート

セレクションソートでは、数字の列から最大の数を探してきて $n$ 番目の数字と入れ替えます。次に先頭から $n-1$ 番目の数字の中から最大の数を探して出して、$n-1$ 番目の数字と入れ替えます。以下同様に繰り返すとソートを実現できます。

セレクションソートを実際に行った例を以下に示します。最初の数字の列は式 (3) で与えられるとします。まず、最大の数字を探すと 8 であることがわかるので最後の数字である 6 と入れ替えます。

$$5,4,6,1,3,2,7,8 \tag{22}$$

次に先頭から 7 番目までの数字の中から最大のものを選んできます。ここでは 7 で 7 番目にあるので特に入れ替えは行いません。

$$5,4,6,1,3,2,7,8 \tag{23}$$

先頭から 6 番目までの数字の中から最大の数字を選ぶと 6 なので 6 番目の数字である 2 と入れ替えます。

$$5,4,2,1,3,6,7,8 \tag{24}$$

以下同様にして行うと

$$3,4,2,1,5,6,7,8 \tag{25}$$
$$3,1,2,4,5,6,7,8 \tag{26}$$
$$2,1,3,4,5,6,7,8 \tag{27}$$
$$1,2,3,4,5,6,7,8 \tag{28}$$
$$1,2,3,4,5,6,7,8 \tag{29}$$

となり、最終的にソートされた数字の列が得られます。

セレクションソートを疑似コードを使って書くと以下のようになります。

**Algorithm 2** selection sort

1: **for** $i=0; i<n; ++i$ **do**
2: 　0 $n-i-1$ までの中から最大のものを探して $n-i-1$ 番目の数字と入れ替える。
3: **end for**

このアルゴリズムは for ループの繰り返しの中で最大値を探すために $n-i-2$ 回の比較が必要です。そのためアルゴリズムの計算量として $\mathcal{O}(n^2)$ となる。

## 2.3 インサーションソート

インサーションソートではソートされていない部分から要素と取り出してソートされた場所の適切な位置に挿入するアルゴリズムです。挿入するときには、挿入する位置より右側のデータを全て一つずつずらす必要があります。

以下に具体的に見てみます。再び最初の数字の列は式 (3) で与えられるとします。最初に一番先頭の数字 5 はすでに 1 つのソートされた列だとします。このとき未ソートの領域 (2 番目から n 番目) から数字を選んでソートされた領域の適切な位置に挿入します。ここでは 2 番目の数字 4 を選んで挿入します。

$$4, 5, |8, 1, 3, 2, 7, 6 \tag{30}$$

列の中の | より左がソート済み領域、右が未ソート領域となっています。ここまでで、先頭から 2 番目までソートされた領域になります。次に 3 番目以降の未ソートの領域から数字をとってきます。ここで 3 番目の数字 8 をとってきます。これをソート領域に挿入すると挿入位置は 3 番目なので特に並び替えは起こりません。

$$4, 5, 8, |1, 3, 2, 7, 6 \tag{31}$$

次に未ソート領域内の 4 番目の数字をソート済み領域に挿入します。

$$1, 4, 5, 8, |3, 2, 7, 6 \tag{32}$$

以下同様に繰り返すと

$$1, 3, 4, 5, 8, |2, 7, 6 \tag{33}$$

$$1, 2, 3, 4, 5, 8, |7, 6 \tag{34}$$

$$1, 2, 3, 4, 5, 7, 8, |6 \tag{35}$$

$$1, 2, 3, 4, 5, 6, 7, 8| \tag{36}$$

となり最終的に全ての領域がソートされた列となります。

インサーションソートを疑似コードで書くと以下のとおりです。

**Algorithm 3** insertion sort

```
for i = 1; i < n; ++i do
    x = data[i]
    j = i
    while j − 1 > 0 and data[j − 1] > x do
        data[j] = data[j − 1]
        j = j − 1
    end while
    data[j] = x
end for
```

インサーションソートの計算時間を考えるために最悪の場合として入力が数字が大きい順に並んでいるとします。このときには、ソート済み領域に次の数字を挿入するたびに全ての列を一つずつずらしていく必要があるため、計算量は $\mathcal{O}(n^2)$ となります。一方で、入力がほとんど小さい順番に並んでいるときには列をずらす操作がほとんど不要となり、結局 n 回の数字の比較で済みます。ゆえにその場合には計算時間は $\mathcal{O}(n)$ となります。

## 2.4 シェルソート

バブルソートやインサーションソートはデータを一つずつ移動していくため効率が悪くなります。そこで $h > 1$ を適当に決めて $h$ ずつ飛び飛びで比較してデータの交換を行う方法が考えられます。これを $h$-整列といいます。

いま例えば、式 (3) に対して $h = 2$ とした $h$-整列を行ってみます。このときには数字を一つ跳びごとに選んで小さな列を作ります。それぞれの列は以下のようになります。

$$5, \ , 8, \ , 3, \ , 7, \tag{37}$$

$$, 4, \ , 1, \ , 2, \ , 6 \tag{38}$$

このそれぞれの列をソートします。ここではインサーションソートをします。まず一つ目の列については

$$5, | \ , 8, \ , 3, \ , 7, \tag{39}$$

$$5, \ , 8, | \ , 3, \ , 7, \tag{40}$$

$$3, \ , 5, \ , 8, | \ , 7, \tag{41}$$

$$3, \ , 5, \ , 7, \ , 8, | \tag{42}$$

を得ます。また二つ目の列については

$$4, | \ , 1, \ , 2, \ , 6 \tag{43}$$

$$1, \ , 4, | \ , 2, \ , 6 \tag{44}$$

$$1, \ , 2, \ , 4, | \ , 6 \tag{45}$$

$$1, \ , 2, \ , 4, \ , 6| \tag{46}$$

$$\tag{47}$$

となります。この二つを合わせると

$$3, 1, 5, 2, 7, 4, 8, 6 \tag{48}$$

を得ます。これが $h(= 2)$-整列です。

シェルソートでは $h$ の値を変えながら $h$-整列を行うソートアルゴリズムです。効率を上げるには初めは $h$ の値を大きく取り、数字を大きく移動させていくことが大切です。以下では $h = \mathrm{int}(n/2)$ から始めて $h$ を順番に半分して小さくするように $h$ を選んで具体的にシェルソートを確認してみます。初期の列を式 (3) として $n = 8$ なので、最初は $h = 4$ です。$h = 4$ で $h$-整列を行うと

$$3, 4, 8, 1, 5, 2, 7, 6 \tag{49}$$

$$3, 2, 8, 1, 5, 4, 7, 6 \tag{50}$$

$$3, 2, 7, 1, 5, 4, 8, 6 \tag{51}$$

となります。次に $h = 2$ として

$$3, 2, 7, 1, 5, 4, 8, 6 \tag{52}$$

$$3, 1, 7, 2, 5, 4, 8, 6 \tag{53}$$

$$3, 1, 5, 2, 7, 4, 8, 6 \tag{54}$$

となります。最後に $h = 1$ として

$$1, 3, 5, 2, 7, 4, 8, 6 \tag{55}$$

$$1, 2, 3, 5, 7, 4, 8, 6 \tag{56}$$

$$1, 2, 3, 4, 5, 7, 8, 6 \tag{57}$$

$$1, 2, 3, 4, 5, 6, 7, 8 \tag{58}$$

を得ます。以上のシェルソートのアルゴリズムを疑似コードを示します。

**Algorithm 4** shell sort

```
1: g = int(n/2)
2: while g > 0 do
3:    for i = g, g < n do
4:       t = data[i]
5:       j = i
6:       while j ≥ g & data[j − g] > t do
7:          data[j] = data[j − g]
8:          j = j − g
9:       end while
10:      data[j] = t
11:   end for
12:   g = int(g/2)
13: end while
14: data[j] = x
```

この疑似コードでは 3 行目から 1 1 行目まででインサーションソートを $g$ ごとに行っています。シェルソートの計算量は $\mathcal{O}(n \log n)$ 程度であるようです。

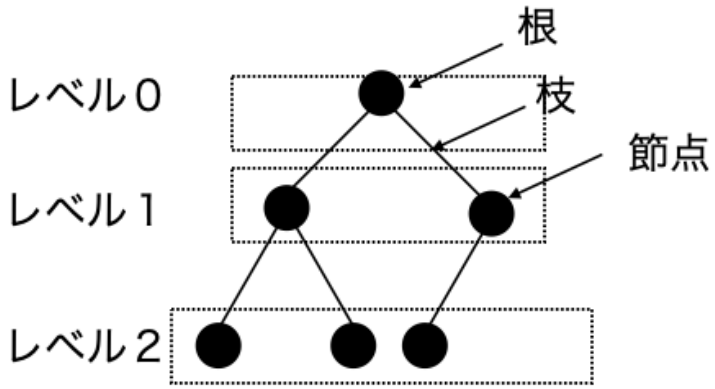
## 2.5 ヒープソート

次にヒープソートについて説明します。ヒープソートはヒープと呼ばれるデータ構造を用いてソートを行う方法です。ここではまずヒープについて説明します。

### 2.5.1 ヒープとは

ヒープは木構造と呼ばれるデータ構造の一種です。木構造は、いくつかの節点と節点を結ぶ枝から構成されています。節点はデータを蓄える場所で枝はデータ間の関係を表しています。木構造には根と呼ばれる節点が一つあり、根から枝が伸びる形て木構造が作られていく。二つの節点 u と v が枝で繋がっていて u が v よりも根に近いとき u は v の親とよび、v は u の子と呼びます。根以外の節点は必ず親を 1 つ持ちます。どの節点も高々k 個の子しか持たない木構造を k 分木と呼び、特に k=2 を 2 分木と呼びます。根から最短で k 本の枝を通って節点に行けるとき、その節点はレベル k であるといいます。根はレベル 0 とします。二分木の場合レベル k にある節点の数はおよそ $2^k$ 個となります。よって最大レベルが k の木構造には $2^{k+1} - 1$ 個の節点が存在します。

2 分木は例えば以下のような構造になります。



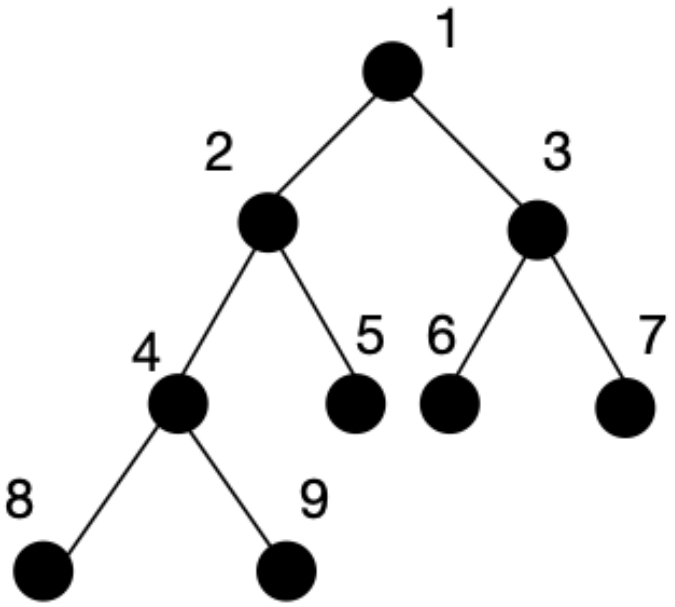

この例ではレベルは２までとなります。

ヒープでは２分木を考え、特にデータの最大値を探しやすいようなデータ構造とします。のちに詳しく説明しますが、ヒープでは最大値を根の位置に割り当てます。データを追加する際には追加したいデータと根を比較して大きな方を根に残します。小さかった方は保持して左か右の子に移動して、同様のことを繰り返します。このとき、根から見て左右の木構造ができるだけバランスよく構成されていた方が便利です。そこでヒープではデータ数が $n$ のときは $k$

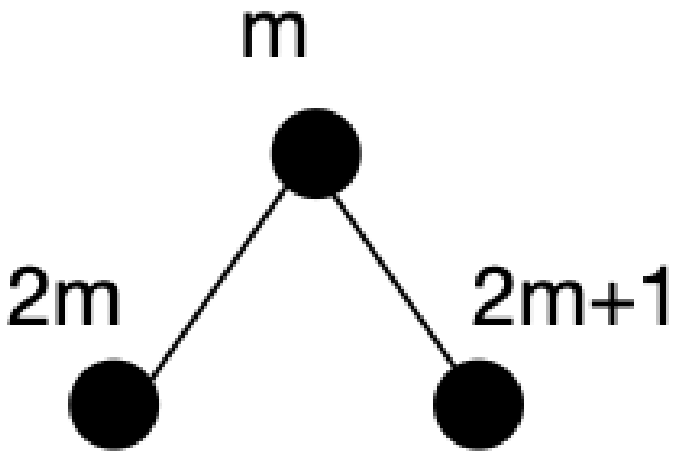
$$2^k - 1 < n \leq 2^{k+1} - 1 \tag{59}$$

を満たす自然数として、レベル $k-1$ までの全ての節点にデータを格納したのち、残りのデータをレベル $k$ に格納することで木構造のバランスを保つようにします。さらにレベル $k$ にデータを追加するときも最も左の節点からデータを追加していきます。

このルールの下でヒープのかく節点に１から番号をつけていきます。まず根には１の番号を振ります。次にレベル１に移動し左から順に２、３の番号を振ります。レベル２ではレベル１の２の節点の子の左から４、５を割り振り次にレベル１の３の節点の子の左から６、７の番号を振ります。以下同様に番号を振っていきます。

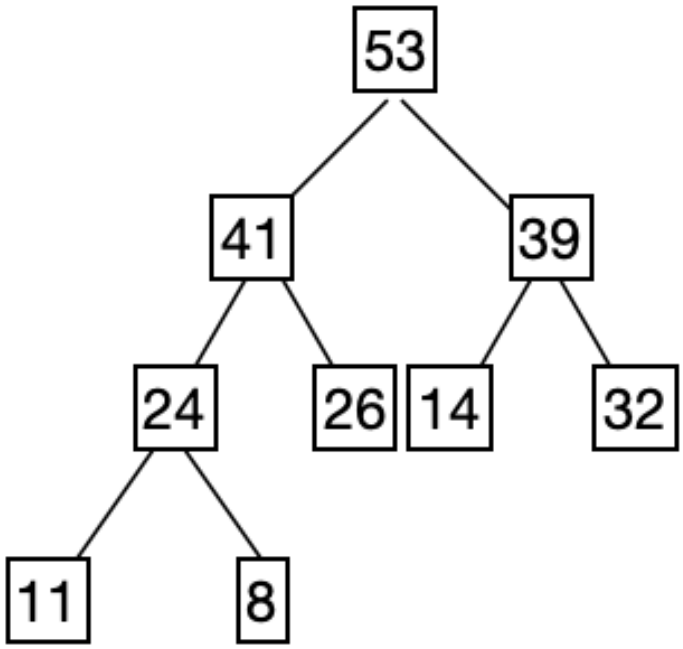


今、一番大きなレベルの節点には左からデータを蓄えているので、このような番号づけで一意に節点を区別できます。またこの番号付だと m 番目の節点の子の番号は $2m$ と $2m+1$ で与えられることがわかります。




この番号付をプログラムの配列のラベルと同一視することで、配列を用いて木構造を表現することができます。

次に節点にデータを蓄える時のルールについて説明します。ヒープではどの枝に対しても、親には子よりも大きなデータを保存するようにします。このルールにより根は必ずデータの最大値となります。




ここまでの議論からデータ数が n のヒープの最大レベルは $[\log_2(n+1)]-1$ で与えられることがわかります。以上のルールをまとめます。

1. 根には番号 1 を割り当てます。

2. 番号 $m$ の左は $2m$ で、右は $2m+1$ とします。

3. データ数 $n$ を超える番号を持つ節点は存在しません。

4. どの枝に対しても、親は子よりも大きな数字を蓄えます。

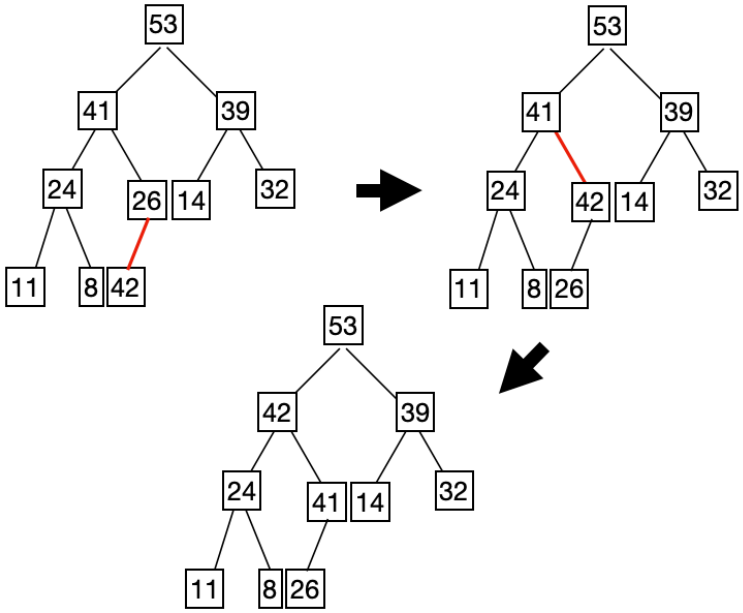
### 2.5.2 ヒープへのデータの格納

今 n 個のデータを蓄えているヒープがあるとして、そこに $n+1$ 個目のデータを追加する方法を考えます。まず、ヒープの木構造から $n+1$ 個目のデータを追加したいときに増えるべき節点の位置は一意に決まります。$n+1$ が偶数なら新たに追加する節点は $\frac{n+1}{2}$ の左側の子、$n+1$ が奇数なら新たに追加する節点は $\frac{n}{2}$ の右側の子となります。親の番号はどちらの場合もまとめて $[\frac{n+1}{2}]$ となります。

$n+1$ 番目のデータの追加は次のように行います。以下では data という配列でヒープを表すことにます。まず、追加したいデータを一時的に $n+1$ 番目の節点におきます。この時、追加した数字が親の数字より大きいときにはヒープのルールを満たしません。そこで他との大小関係を確認する必要があります。親のデータは $\mathrm{data}[[\frac{n+1}{2}]]$ で与えられます。もし追加したデータが親のデータより大きい場合、すなわち

$$\mathrm{data}[n+1] > \mathrm{data}[[\frac{n+1}{2}]] \tag{60}$$

の時には親のデータと交換します。交換したら、交換した先でさらに親のデータと大小関係を比較し親のデータが小さければ交換します。この作業を繰り返し子が親より小さなデータを蓄えるようになったらデータの追加が完了です。以下に 42 の数字を追加するときのヒープの変化を示します。




赤色の線が子と親の代償関係が不整合になっている箇所です。そこの場所ではこのデータ (42) と親のデータ (26) を入れ替えます。入れ替えた先でさらに親のデータ (41) と追加データ (42) の大小関係が不整合なのでデータを入れ替えます。最後には親と子の代償関係が整合するヒープが構成できました。

このヒープに対するデータの追加のアルゴリズムを疑似コード書くと以下のようになります。(配列のラベルは 0 から始めることに注意してください。)

**Algorithm 5** push heap

1: $x$ : 追加したいデータ
2: $n$: 追加する前のヒープが蓄えているデータ数
3: $n++$
4: data[n-1] = x
5: $i = n$
6: $j = \mathrm{int}(i/2)$
7: **while** $j > 0$ & $x >$ data$[j-1]$ **do**
8: data[i-1]=data[j-1]
9: $i = j$
10: $j = \mathrm{int}(i/2)$
11: **end while**
12: data$[i-1] = x$

### 2.5.3 最大データの取り出し

次にヒープから最大の数を取り出す方法について説明します。最大データは根にあります。問題は根のデータを削除した後に如何にしてヒープを再構成するのかです。ヒープの根のデータを取り出すと、ヒープに蓄えられているデータの数が一つ減ります。ヒープを再構成するために、まず一時的にヒープの末尾のデータを根に移動させます。これによってヒープの木構造は保たれます。しかし、末尾のデータは一般に小さな数字なのでそれを根に移動すると親と子の大小関係が満たされなくなります。そこで親と子の大きさを比較してどちらかの子が親より大きかったら、データの移動が必要になります。以下にヒープから最大値を取り出してヒープを再構成するアルゴリズムを示します。

**Algorithm 6** delete heap

$n$: 削除する前のヒープが蓄えているデータ数
$x = \mathrm{data}[0]$
$n--$
$\mathrm{data}[0] = \mathrm{data}[n-1]$
$i = 1$
**while** $2i \leq n$ **do**
  $j = 2i$
  **if** $2i+1 \leq n$ & $\mathrm{data}[2i-1] < \mathrm{data}[2i]$ **then**
    $j = 2i+1$
  **end if**
  **if** $\mathrm{data}[i-1] \geq \mathrm{data}[j-1]$ **then**
    break
  **else**
    $t = \mathrm{data}[i-1]$
    $\mathrm{data}[i-1] = \mathrm{data}[j-1]$
    $\mathrm{data}[j-1] = t$
  **end if**
  $i = j$
**end while**
return $x$

### 2.5.4 ヒープソート

以上のヒープ構造を用いてソートを行うがヒープソートです。ヒープソートでは最初にヒープにデータを追加するアルゴリズムを用いて、データを順に追加していきます。全てのデータを追加した後に、順番に大きい数字から取り出していきます。それでは実際に上で定義した push_heap と delete_heap を用いてソートをしてみます。入力された配列は式 (3) で与えられるとして、順番にヒープに追加していきます。最初は 5 を根に割り当てます。

$$5, |N, N, |N, N, N, N, |N \tag{61}$$

今回はデータ数は 8 であることがわかっているので、配列のサイズを 8 で固定しておいて挿入されていない節点については $N$ と表記してあります。またレベルがわかりやすいようにレベルが変わる箇所で | を挿入しています。次に 4 を追加すると

$$5, |4, N, |N, N, N, N, |N \tag{62}$$

となります。次に追加するのは 8 です。先のアルゴリズムによるとまず一時的に 5 を蓄えている節点の右の子に 8 を置きます。その後に親との大小関係を確認すると子 (8) の方が親 (5) よりも大きな数字なので親と子を入れ替えて、以下を得ます。

$$5, |4, 8, |N, N, N, N, |N \tag{63}$$

$$8, |4, 5, |N, N, N, N, |N \tag{64}$$

1, 3, 2 を追加しますが、こちらは親と子の大小関係は問題にならずに追加できます。

$$8, |4, 5, |1, N, N, N, |N \tag{65}$$

$$8, |4, 5, |1, 3, N, N, |N \tag{66}$$

$$8, |4, 5, |1, 3, 2, N, |N \tag{67}$$

7 の追加ではまず 5 を蓄えている節点の右の子に 7 を入れますが、親と子の大小関係を満たしてないので、5 と 7 を入れ替えます。

$$8, |4, 5, |1, 3, 2, 7, |N \tag{68}$$

$$8, |4, 7, |1, 3, 2, 5, |N \tag{69}$$

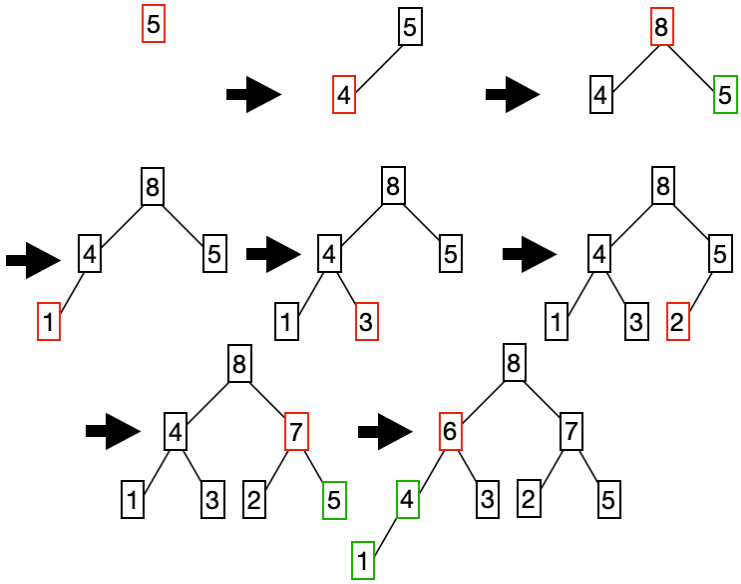
最後に 6 を 1 の節点の左の子として追加しますが、親と子の大小関係を満たすように 2 回データの入れ替えを行います。

$$8, |4, 7, |1, 3, 2, 5, |6 \tag{70}$$

$$8, |4, 7, |6, 3, 2, 5, |1 \tag{71}$$

$$8, |6, 7, |4, 3, 2, 5, |1 \tag{72}$$

これでヒープが完成します。以下にここまでの流れを図で示しています。赤色の枠が新しく追加した数字で、緑色の枠がデータが追加されたことで移動したデータとなります。




次に得られたヒープから最大のデータを順番に取り出していきます。ヒープを配列で表現すると最大の数は必ず一番左になるのでまず、一番左のデータ（ヒープの根のデータ）を取り出します。取り出すと根にデータがなくな理、ヒープ構造を保っていないので配列の末尾のダータを先頭に持っていき親と子の大小関係をみながらヒープを作成しなおします。この操作を実際に見ていきます。いま先ほどのヒープがあるとします。

$$8, |6, 7, |4, 3, 2, 5, |1 \tag{73}$$

ここから 8 を取り出して末尾の 1 の数字を根に移動します。すると根の部分で親と子の大小関係が崩れてしまうので、ヒープを保つためにデータを入れ替えます。入れ替える時には左右の子のデータを比較

して大きな数字を持つ子と親を入れ替えます。今回の場合には根に 1 があり子の左が 6 で右が 7 なので、7 と 1 を入れ替えます。すると今度は入れ替えた先で親（1）と子（2,5）で大小関係を満たさないので 1 と 5 を入れ替えてヒープができました。

$$1, |6, 7, |4, 3, 2, 5, |N \tag{74}$$

$$7, |6, 1, |4, 3, 2, 5, |N \tag{75}$$

$$7, |6, 5, |4, 3, 2, 1, |N \tag{76}$$

次に 7 を取り出して再び末尾のデータの 1 を根に移動して、親子の大小関係を見ながらヒープを作り直していきます。

$$1, |6, 5, |4, 3, 2, N, |N \tag{77}$$

$$6, |1, 5, |4, 3, 2, N, |N \tag{78}$$

$$6, |4, 5, |1, 3, 2, N, |N \tag{79}$$

以上同様に繰り返します。途中で作られるヒープだけ見ていくと以下のようは変遷をたどります。

$$5, 4, 2, 1, 3, N, N, N \tag{80}$$

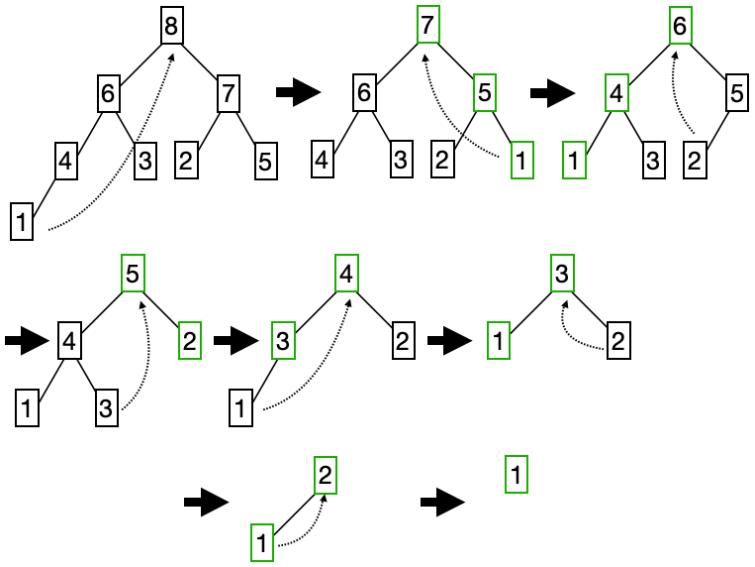
$$4, 3, 2, 1, N, N, N, N \tag{81}$$

$$3, 1, 2, N, N, N, N, N \tag{82}$$

$$2, 1, N, N, N, N, N, N \tag{83}$$

$$1, N, N, N, N, N, N, N \tag{84}$$

この最大値を順に取り出していく流れを以下に図に示します。




これらの手続きを疑似コードで書くと以下の通りです。

**Algorithm 7** heap sort

```
data: ソートしたい入力データ
n:入力データの配列の長さ
for i = 0; i < n; ++i do
    push_heap(data[i])
end for
for i = 0; i < n; ++i do
    data[n-i-1] = deleta_heap()
end for
```

ヒープは木構造の均衡が保たれているのでデータの追加と取り出しは $\mathcal{O}(\log(n))$ の計算量で済みます。故に全体としては $\mathcal{O}(n\log(n))$ の計算量で完了します。

### 2.5.5 上記ヒープソートの改良

ここまでのヒープソートでは入力データとは別にヒープを蓄える配列を一時的に用意してヒープソートを行ってきました。しかし実はヒープではデータを蓄えていれる配列をそのままヒープとして扱うことができます。配列の k 番目のデータをヒープに入れる時には、配列の 0 から $k-2$ 番目までをヒープとみなして操作をすることでそれを実現できます。データを取り出すときも、取り出したデータを右から詰めていことで、配列をヒープとして扱うことができます。

またデータを順にヒープに追加してヒープを構成する方法についても改良することができます。まず親子の大小関係を気にしないで与えられた配列が二分木を表しているとします。親子関係を満たすように配列の末尾から親子関係を確認していきます。配列の末尾から順番に接点を見ていき子を持つ節点を探します。この節点の子には配列の末尾のデータが含まれます。もしその節点が子を一つ（末尾のデータ）しか持たないなら親と子の大小関係を比較して親が子よりも小さな数字の場合データを入れ替えます。もし節点が子を二つ持つ場合には子のうち大きな数字を持つ方と親を比較して親の方が小さければ、二つを入れ替えます。この操作を配列の末尾から始めて子を持つ節点に対して行っていきます。この操作を繰り返していくと、対象としている節点の子が子を持つようなケースも現れます。その場合にも、まず接点のデータと子のうち大きなデータを持つ子とを比較して、親の方が子よりも小さな値の場合にはデータを入れ替えます。入れ替えた後に子と子の子と大小関係を比較し、必要があればデータを入れ替えます。以下この操作を大小関係が満たされるまで進めていきます。以上の操作を繰り返すことで最終的にヒープ構造を作ることができます。この方法は、木構造の下の階層（高いレベル）から構造を見ていって部分木を整形してヒープを少しずつ作っていっています。

具体的にこの操作を式 (3) から始めてみていきます。まず最初の配列を二分木とみなします。

$$5, |4, 8, |1, 3, 2, 7, |6 \tag{85}$$

| は前回同様レベルが変わる箇所に挿入してあります。末尾から子を持つ節点を探します。ここでは 1 が格納されています。（実際にはこの操作は配列の長さを $n$ として $\mathrm{int}(n/2)$ 番目の配列になります。）1 が格納されている節点の子は 6 で親子の大小関係を満たしていないので 1 と 6 を入れ替えます。

$$5, |4, 8, |6, 3, 2, 7, |1 \tag{86}$$

次に親になっている節点は 8 が格納されているますが、こちらの子（2 と 7）は親子の大小関係を満たしているので、そのままです。次の親は 4 が格納されていますが、子（3,6）と大小関係を満たしていませ

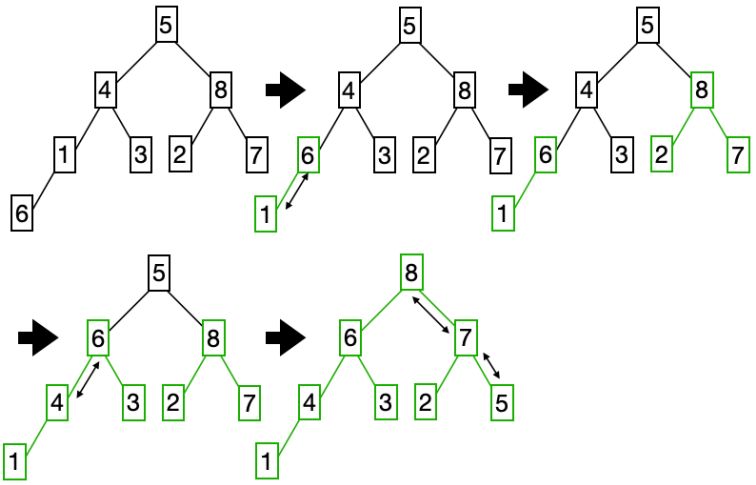
ん。そこで子のうち大きな数字 6 と 4 を入れ替えます。入れ替えた後、子（新しく 4 が格納された節点）とその子をみてこちらは大小関係に問題ありません。

$$5, |6, 8, |4, 3, 2, 7, |1 \tag{87}$$

次の親は根（５が格納）ですが子（６と８）と大小関係を満たしていないので、子のうち大きな数字（８）とデータを交換します。交換した先でも親（５）と子（２、７）が親子の大小関係を満たしていないのでデータ交換（５と７）を行って終了です。

$$8, |6, 7, |4, 3, 2, 5, |1 \tag{88}$$

以上の操作を木構造を用いて表すと以下のようになります。




緑色の節点と枝が整形が完了した箇所になります。改良されたヒープソートのアルゴリズムの疑似コードを以下にまとめます。

**Algorithm 8** heap sort 2

```
1: data: ソートしたい入力データ
2: n:入力データの配列の長さ
3: for k = int(n/2); k > 0; − − k do
4:    i = k
5:    j = 2i
6:    x = data[i − 1]
7:    while j ≤ n do
8:       if j < n&data[j] > data[j − 1] then
9:          j = j + 1
10:      end if
11:      if x < data[j − 1] then
12:         data[i − 1] = data[j − 1]
13:         i = j
14:         j = 2i
15:      else
16:         break
17:      end if
18:   end while
19:   data[i − 1] = x
20: end for
21: for k = n − 1; 0 < k − 1; − − k do
22:   x = data[k]
23:   data[k] = data[0]
24:   i = 1
25:   j = 2
26:   while j ≤ k do
27:      if j < k&data[j] > data[j − 1] then
28:         j = j + 1
29:      end if
30:      if x < data[j − 1] then
31:         data[i − 1] = data[j − 1]
32:         i = j
33:         j = 2i
34:      else
35:         break
36:      end if
37:   end while
38:   data[i − 1] = x
39: end for
```

この疑似コードでは、3 行目のループでヒープを作成し 2 1 行目でデータを取り出しています。このアルゴリズムではヒープを作成する箇所の計算量を推定します。簡単のため $n = 2^k - 1$ をデータ数とします。この時レベル $k$ が最高レベルとなります。レベル $j(\leq k)$ における節点を根とした部分木を作るた

めには部分木の高さが $k-j$ なのでデータの比較としてはたかだか $2(k-j)$ 程度です。レベル $j$ 除ける接点の数は $2^{j-1}$ なので計算量としては

$$\sum_{j=1}^{k-1} 2(k-j)2^{j-1} = 2^{k+1} - 2k - 2 < 2n \tag{89}$$

となり $\mathcal{O}(n)$ 以下に抑えられることがわかります。

## 2.6 クイックソート

クイックソートでは与えられた配列の中から適当な値 $x$ を持ってきて、$x$ より小さいか、$x$ に等しいか、$x$ より大きいかで配列を分割します。この適当に選んだ値を pivot と呼びます。pivot より小さい配列と pivot より大きい配列をさらに別の数字を持ってきて分割するというのを再帰的に繰り返してソートを行います。入力配列を data としたときに、クイックソートの流れは以下の通りです。

1. data の長さが 1 なら data をリターンします。
2. data から任意に 1 つ要素 $x$（pivot）を取ります。
3. data の各要素と $x$ を比較して $x$ より小さい集合を data1、$x$ と等しい要素を data2、$x$ より大きい集合を data3 として分割します。
4. data1,data3 を再帰的にクイックソートします。
5. data1,data2,data3 の順に連結した配列をリターンします。

クイックソートの基本的なアルゴリズムは以上の通りです。

pivot の選び方はさまざまな方法が考えられ、その方法によってクイックソートの計算効率が変わります。pivot 選択の考えられる方法としては（１）配列の先頭の数字（２）配列の末尾の数字（３）配列の先頭、末尾、中央を大きい順に並べた時の中央の値（４）配列の中の要素からランダムに選択する、などが考えられます。pivot 選択の良し悪しがクイックソートの計算量にどのように影響するかを具体的に確認してみます。もし pivot として選んだ数字がたまたま配列の中の最小値だった場合、分割後には data1 は空で data2 と data3 にデータが入ることになります。場合によりますが、最小値を取る要素がいくつもない場合には、data3 の配列にほとんどのデータが集中することになってしまいます。これが繰り返されると結局 $\mathcal{O}(n)$ かいの分割操作が繰り返されので、大小の比較の回数がそれぞれの分割に対して要素数程度の計算量がかかるので全体として $\mathcal{O}(n^2)$ 程度の計算量がかかります。一方で pivot の選択が入力データの中央値にあれば分割後の配列 data1 と data3 はほとんど均等な要素数を持つので繰り返し回数は $\mathcal{O}(long(n)$ となり全体の計算量としては $\mathcal{O}(n\log(n))$ 程度となります。

そのため pivot の選び方で計算効率が大きく変わります。もし仮にほとんどソートされている配列で、配列の先頭や末尾を pivot として選んでしまうと、pivot が配列の最小値や最大値に近い値となってしまい、計算効率が悪くなってしまいます。入力データの性質があらかじめある程度わかっていればそれに応じた pivot の選択ができますが、一般の場合に最適な選択をするのは難しそうです。

クイックソートを実際に実行するために、配列の先頭を pivot とする方法で式 (3) に対してクイックソートを実行してみます。最初は pivot は 5 なので配列の中で 5 より小さい要素と大きい要素を分けます。

$$[4,1,3,2],[5],[8,7,6] \tag{90}$$

さらに各配列に対して同様のことを行います。最初の配列に対しては 4 を pivot として

$$[1,3,2][4][] \tag{91}$$

を得ます。4 より大きな数字はないので、3 番目の配列は空になっています。後から疑似コードで示すように空の配列が与えられた処理もしっかりと記述しないと動くコードにはなりません。さらに $[1,3,2]$ に対して行うと

$$[], [1], [3,2] \tag{92}$$

をえて、最後に $[3,2]$ に対して

$$[2], [3], [] \tag{93}$$

を得ます。これで $[4,1,3,2]$ に対しては処理が終了しました。$[8,7,6]$ に対しても同様に処理を行うことができます。

先ほど述べたように pivot の選択はクイックソートの計算量を左右するため重要です。しかし、事前に入力データの性質がわかっていなければ適切な pivot の選択は難しい問題です。そのため（４）の乱数を使って配列の中の要素から選択する方法が一番バランスが良いとも言えます。アルゴリズムの実行に乱数を含むものを乱択アルゴリズムと言います。以下では（４）の pivot 選択の方法を用いたクイックソートの疑似コードを示します。

**Algorithm 9** quick sort

```
1: data: 入力配列
2: n:data の配列の大きさ
3: if n == 0 then
4:     return data
5: end if
6: r:0 から n − 1 までの乱数
7: p = data[r]
8: la = []
9: ra = []
10: pa = []
11: for e in data do
12:     if e < p then
13:         la.append(e)
14:     else
15:         if p < e then
16:             ra.append(e)
17:         else
18:             pa.append(e)
19:         end if
20:     end if
21: end for
22: return quick_sort(la) + pa + quick_sort(ra)
```

３、４行目に入力配列が空の場合の処理が書かれています。クイックソートのような再帰的に関数が呼び出されるアルゴリズムの場合には、こうした処理を正しく書いておかないと無限ループに入ってしまうなどバグの原因となるので注意してください。

## 2.7 マージソート

マージソートではソートされた二つの配列をソートされた一つの配列にマージするアルゴリズムを用いたソートです。まずソートされた二つの配列から一つのソートされた配列を作るアルゴリズムについて考えます。今配列として配列 $A, B$ があるとします。$A, B$ はともにソートされているとします。マージ後の配列として空の配列 $C$ を用意しておきます。また配列 $A, B$ の要素をさすポインタを $a, b$ としてはじめに $a, b$ は $A, B$ の先頭を指しているとします。はじめに $A$ と $B$ の配列の $a, b$ が指している要素（配列の先頭）を見て小さな値の方を選びます。今仮に $A$ の先頭の方が $B$ の先頭より小さかったとします。するとまず $A$ の先頭の数字を $C$ に追加します。追加後に $a$ を一つ進めます。一つ進めた後再び $a$ と $b$ の指している要素を比較して配列 $C$ に追加します。追加した方の配列のポインタを一つ進めて、どちらかのポインタが配列の末端に来たら、残ったもう片方の配列の要素を全部 $C$ に追加します。このようにして作られた配列 $C$ は配列 $A, B$ の要素を二つ持つソートされた配列となります。このアルゴリズムを疑似コードで書くと以下のようになります。

**Algorithm 10** merge array

```
A: ソートされた入力配列
n_A:配列 A の長さ
B:ソートされた入力配列
n_B:配列 B の長さ
C:ソート後の配列（最初は空）
a = 0
b = 0
while a < n_A&b < n_B do
    if A[a] < B[b] then
        C.append(A[a])
        a++
    else
        C.append(B[b])
        b++
    end if
end while
if a == n_A then
    for i = b; i < n_B; ++i do
        C.append(B[i])
    end for
else
    for i = a; i < n_A; ++i do
        C.append(A[i])
    end for
end if
return C
```

この mearge_array のアルゴリズムを使って配列のソートを考えるます。要素が一つだけの配列は自明にソートされているとみなすことができるので与えられた配列を要素一つの配列に分割して順番に merge_array を実行していけば良いことがわかります。そこでソートしたいデータを分割して配列の配列

を作って、その要素である配列を順にマージしていくことでソートができると考えられます。このアルゴリズムを疑似コートで書くと以下の通りです。

**Algorithm 11** step

```
arr: 配列の配列
n:arr の長さ
ret = []
if n%2 == 0 then
    for i = 0; i < n; i = i + 2 do
        ret.append(merge_array(arr[i],arr[i+1]))
    end for
else
    for i = 0; i < n − 1; i = i + 2 do
        ret.append(merge_array(arr[i],arr[i+1]))
    end for
    ret.append(arr[n-1])
end if
return ret
```

この step という関数は arr という配列の配列が与えられた時、二つずつ merge_array に従ってマージをしていきます。この step を繰り返し呼ぶことで最終的にソートされた配列が得られます。

**Algorithm 12** merge_sort

```
data:入力配列
n:data の長さ
ret = [] //配列の配列を格納する
for i = 0; i < n; + + i do
    ret.append([data[i]])
end for
while size of ret[0] != n do
    ret = step(ret)
end while
return ret
```

このアルゴリズムを実際に式 (3) に対して行なっています。まず要素を分割して 1 要素の配列の配列を作成します。

$$[[5],[4],[8],[1],[3],[2],[7],[6]] \tag{94}$$

1 つおきに merge_array を使ってマージを行なっていきます。最初のマージで

$$[[4,5],[1,8],[2,3],[6,7]] \tag{95}$$

となり、次のマージで

$$[[1,4,5,8],[2,3,6,7]] \tag{96}$$

を得ます。もう一度マージを行うと所望の結果を得ることができます。このアルゴリズムでは、マージ後の配列の配列がほとんど同じ長さの配列で構成されることで計算効率が上がっています。マージソートは 2 分木として書くこともでき、その深さは $\mathcal{O}(\log(n))$ となります。マージはたかだか n 回なので全体としは $\mathcal{O}(n\log(n))$ 程度の計算量となります。

## 3　ソートの計算量について

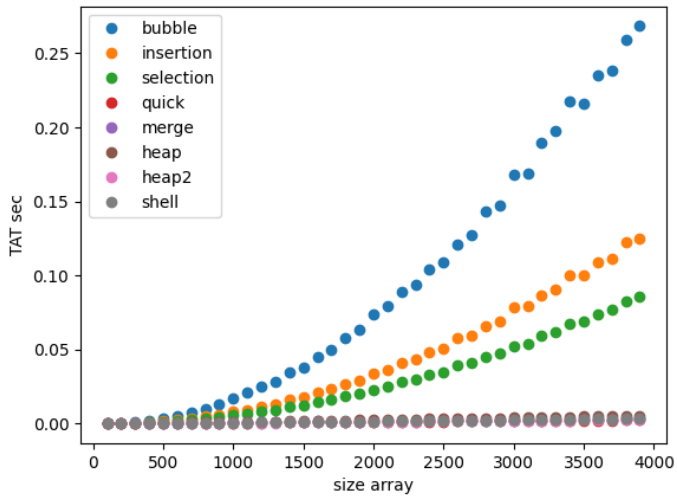
紹介したアルゴリズムの中で最も計算量の少ないアルゴリズムは $\mathcal{O}(n\log(n))$ の計算量でした。実はソートアルゴリズムではこれよりも効率的なアルゴリズムは存在しないことがわかっています。ここではその点について説明をしていきたいと思います。いまソートしたい配列が長さ $n$ で要素は全て異なっているとします。ソートでは二つの要素 $a_i$ と $a_j$ を比較して入れ替えるかどうかが基本的な操作でした。この入れ替えるかどうかの可能性の分岐を 2 分木で表すことができます。ソートのアルゴリズムを決めるとこの比較と対応する二分木が決まります。一方長さ $n$ の配列に対して可能な配列の並びは $n!$ です。そのため。二分木で全ての可能な配列の順序を得ようとすると二分木の深さ $k$ は

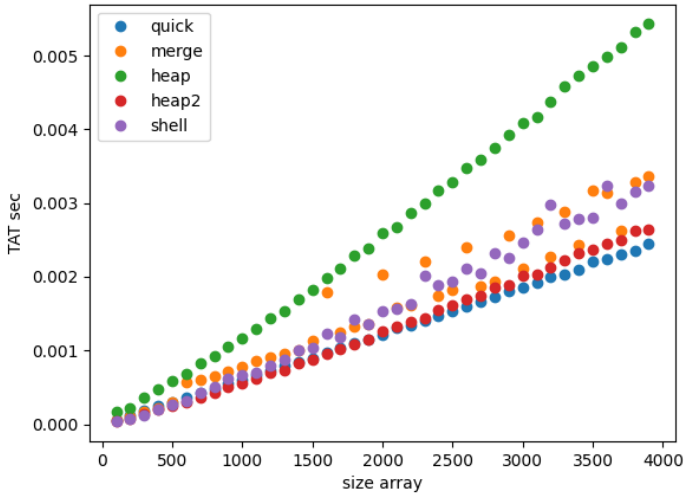
$$k \sim \log n! \tag{97}$$

で与えれ、これはおよそ $\mathcal{O}(n\log(n))$ となります。つまり最低でも $\mathcal{O}(n\log(n))$ の深さの二分木でないと全ての配列の可能性を網羅できません。二分木を辿っていく過程がソートのアルゴリズムの過程に対応しているので結局必要な計算量は $\mathcal{O}(n\log(n))$ として見積もれることがわかります。

## 4　TAT 計測

最後にそれぞれのアルゴリズムの TAT を計測しました。計測では配列の大きさを 100 から 4000 まで 100 間隔で変化させました。それぞれの配列の大きさで 10 回ずつ乱数で配列を生成し、その配列を並び替えて 10 回の TAT の平均をプロットしています。






## 参考文献

[1] 辻真吾、下平英寿、「Python で学ぶアルゴリズムとデータ構造」